Journal of Statistical Software

January 2012, Volume 46, Issue 5. *http://www.jstatsoft.org/*

Multi-Objective Parameter Selection for Classifiers

Christoph Müssel University of Ulm
Ludwig Lausser University of Ulm
Markus Maucher University of Ulm
Hans A. Kestler University of Ulm

Abstract

Setting the free parameters of classifiers to different values can have a profound impact on their performance. For some methods, specialized tuning algorithms have been developed. These approaches mostly tune parameters according to a single criterion, such as the cross-validation error. However, it is sometimes desirable to obtain parameter values that optimize several concurrent – often conflicting – criteria. The **TunePareto** package provides a general and highly customizable framework to select optimal parameters for classifiers according to multiple objectives. Several strategies for sampling and optimizing parameters are supplied. The algorithm determines a set of Pareto-optimal parameter configurations and leaves the ultimate decision on the weighting of objectives to the researcher. Decision support is provided by novel visualization techniques.

Keywords: classification, parameter tuning, multi-objective optimization, R.

1. Introduction

Many state-of-the-art classifiers have free parameters that influence their behavior like generalization performance and robustness to noise. Choosing the proper parameters is an important aspect of adapting classifiers to specific types of data. For example, tuning the cost and kernel parameters of support vector machines is essential for obtaining sensible results (Chang and Lin 2001; Fröhlich and Zell 2005; Pontil and Verri 1998). Specialized parameter tuning approaches for classifiers have been developed. Chapelle *et al.* (2002) optimize support vector machine (SVM) parameters by a gradient descent of different estimates of the generalization error. Various evolutionary algorithms and swarm algorithms have been designed to optimize SVM parameters (e.g., Chunhong and Licheng 2004; Zhang *et al.* 2009; de Souza *et al.* 2006; Kapp *et al.* 2009). Kalos (2005) suggests a particle swarm optimization algorithm to determine the structure of neural networks. Kohavi and John (1995) introduce a general framework that optimizes a classifier according to its cross-validation error in a best-first search algorithm. Sequential parameter optimization (Bartz-Beielstein *et al.* 2005;

Bartz-Beielstein 2006) is another parameter tuning framework that tries to cope with the stochastically disturbed results of search heuristics by repeated evaluations. For this methodology, the R (R Development Core Team 2011) package **SPOT** (Bartz-Beielstein *et al.* 2011) has been developed. Also other R packages include specialized tuning functions for classifiers. The **e1071** package (Dimitriadou *et al.* 2010) provides a generic tuning function that optimizes the classification error for some classifiers, such as k-nearest neighbour, SVMs (Vapnik 1998), decision trees (Breiman *et al.* 1984), and random forests (Breiman 2001). The `tuneRF` function in the **randomForest** package tunes the number of variables per split in a random forest with respect to the out-of-bag error (Liaw and Wiener 2002).

Most frequently, parameters are tuned in such a way that they optimize a single criterion, such as the cross-validation error, which can be a good estimate of the generalization performance. However, it is sometimes desirable to obtain parameter values that optimize several concurrent criteria at the same time. In this context, it is often hard to decide which trade-off of these criteria matches the requirements best. The discipline of multiple criteria decision analysis (MCDA) analyzes formal approaches to support decision makers in the presence of conflicting criteria (see, e.g., Belton and Stewart 2002).

One possibility of optimizing a number of criteria is combining them in a weighted sum of objective functions. Still, this requires the definition of a fixed weighting, which is often highly subjective and sensitive to small changes in these weights (Deb 2004). A more flexible way of optimization is based on the identification of Pareto-optimal solutions (Laux 2005). Here, all objectives are treated separately, which means that there is no strict ordering of solutions. This selection procedure retrieves all optimal trade-offs of the objectives and leaves the subjective process of selecting the desired one to the researcher. To date, parameters are almost always tuned in a single-objective manner. Exceptions are a specialized two-objective genetic algorithm that tunes the parameters of SVMs according to training error and model complexity in terms of number and influence of the support vectors (Igel 2005; Suttorp and Igel 2006). Also Zhang (2008) converts a multi-objective optimization of SVM parameters into a single-objective optimization by introducing weight parameters to control the trade-offs.

We developed the **TunePareto** package for the statistical environment R (R Development Core Team 2011) that allows for a flexible selection of optimal parameters according to multiple objectives. Unlike previously published specialized tuning algorithms, this general approach is able to identify optimal parameters for arbitrary classifiers according to user-defined objective functions. Parallelization support via the snowfall package (Knaus *et al.* 2009) is also included. The software further provides multiple visualization methods to analyze the results. It is freely available from the Comprehensive R Archive Network at `http://CRAN.R-project.org/package=TunePareto`.

2. Multi-objective parameter selection

As briefly mentioned, most existing parameter tuning approaches optimize classifier parameters according to a single objective, which is most often the cross-validation error. However, in some contexts, a classifier may have to meet several criteria. These criteria are usually conflicting, such that optimizing one leads to a deterioration of another. Imagine one would like to determine a predictor for a disease. Important properties of such a classifier are the sensitivity (i.e., the fraction of cases predicted correctly by the classifier) and the specificity

(i.e., the fraction of controls predicted correctly by the predictor). It is somehow intuitive that these objectives usually cannot be optimized at the same time: e.g., a classifier that classifies all examples as cases has a perfect sensitivity, but a worst-case specificity and vice versa. A trade-off of sensitivity and specificity is often the desired result. This raises the question of how to optimize a classifier according to more than one objective. One possibility is to join the objectives in a weighted sum, i.e., $f(c) = w_1 \cdot \mathit{Sens}_d(c) + w_2 \cdot \mathit{Spec}_d(c)$, where $\mathit{Sens}_d(c)$ is the sensitivity of a classifier c on data set d, and $\mathit{Spec}_d(c)$ is the specificity of c. However, it remains unclear how to choose the weights w_1 and w_2 appropriately. Usually, it is not known which weight combination is associated with the desired trade-off. Often, several trade-offs may be valid. Furthermore, weighted sums of objectives cannot retrieve all individually optimal solutions if the optimization problem is non-convex (Deb 2004). As it is generally unknown whether an optimization problem is convex, it is often more desirable to determine optimal classifier parameter configurations according to dominance-based methods. These include multiple trade-offs, which may later be analyzed manually by the user to choose the most appropriate one for a specific scenario. For example, a classical method of choosing a good trade-off would be to test several classifiers and parameters, to plot them in a ROC curve, and to choose a good classifier on the basis of this curve. This is a special case of a multi-objective optimization.

2.1. Pareto optimality

Dominance-based selection techniques provide a means of including all possible trade-offs in the set of solutions (i.e., classifier parameter configurations) by considering the objective functions separately. So-called *Pareto-optimal solutions* are those solutions that cannot be improved in one objective without getting worse in another objective. Here, we do not consider reflexive partial orders such as the Pareto-order (see, e.g., Pappalardo 2008; Luc 2008). The objective function values of Pareto-optimal solutions with different trade-offs form the so-called *Pareto front*. An example is depicted in Figure 1. When there are multiple optimal solutions representing different trade-offs, it is often advisable to leave the final decision of the preferred trade-off to a human expert.

We now introduce a formal definition of the optimization problem (see also Deb 2004). Define a set of classifiers $C = c_1, \ldots, c_n$ with different parameter configurations. The classifiers are rated according to M objective functions $F(c) = (f_1(c), \ldots, f_M(c))$.

- A classifier $c_i \in C$ *dominates* a classifier $c_j \in C$ if for all objective functions f_m, $f_m(c_i) \preceq f_m(c_j), m \in \{1, \ldots, M\}$, and if there is at least one objective $f_{m^*}, m^* \in \{1, \ldots, M\}$ with $f_{m^*}(c_i) \prec f_{m^*}(c_j)$. Here, $\prec$ and $\preceq$ are the general "better" and "better or equal" relations, depending on whether the objectives are maximization or minimization objectives.
- A classifier is called *Pareto-optimal* if there is no classifier in C that dominates it.
- All Pareto-optimal classifiers form the *(first) Pareto-optimal set* $\mathcal{P}_1(C)$. Furthermore, we inductively define the *i-th Pareto-optimal set* $\mathcal{P}_i(C)$ as the Pareto-optimal set among the solutions that are not in one of the preceding Pareto-optimal sets, i.e., the Pareto-optimal solutions of $C \setminus \bigcup_{j=1}^{i-1} \mathcal{P}_j(C)$.
- The *i-th Pareto front* $\mathcal{PF}_i(C) = \{F(c) \mid c \in \mathcal{P}_i(C)\}$ is the set of fitness values of the combinations in the i-th Pareto set.

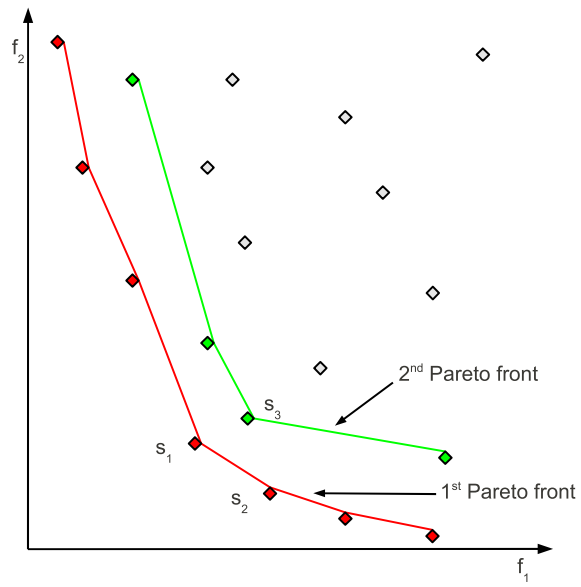

Figure 1: Solutions of two minimization objectives f_1 and f_2. The solutions on the first Pareto front (in red, e.g., s_1 and s_2) are not dominated by any other solution. Solutions on the second front (in green, such as s_3) are dominated by solutions on the first front, but dominate solutions on subsequent fronts.

Dominance imposes a *strict partial order* on the classifier set C:

- It is transitive, i.e., if a classifier c_i dominates another classifier c_j and c_j dominates c_k, then c_i automatically dominates c_k.
- It is not reflexive, i.e., a classifier c_i cannot dominate itself.
- In particular, it is not complete: If c_i does not dominate c_j, c_j does not necessarily dominate c_i, as one classifier may be better in one objective, and the other classifier may be better in another objective.

Hasse diagrams visualize the transitive reduction of a partially ordered set in form of a directed acyclic graph: The elements are the nodes of the graph, and the precedence (dominance) relations are represented by edges. The transitive reduction yields only direct dominance relations, i.e., transitive edges are removed. An example of a Hasse diagram is depicted in Figure 2.

In the case of classifier parameter optimization, several stochastic factors are introduced to the strict partial order: If the ranges of optimized parameters are non-finite (e.g., continuous or unbounded), not all parameter configurations can be evaluated. Sampling strategies and search heuristics often introduce a high amount of randomness, e.g., random sampling approaches or evolutionary algorithms. Furthermore, the objective functions are always approximations of theoretical measures (e.g., the cross-validation error is an estimate of the true classification risk on the fixed, but unknown distribution of the data set). Some classifiers also introduce inherent stochasticity if they involve random decisions (e.g., random forests

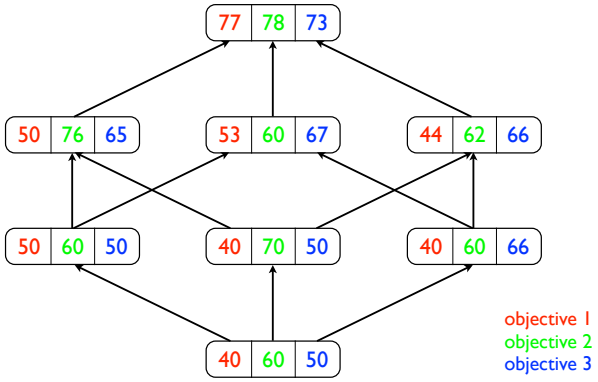

Figure 2: Hasse diagram of a strict partial order according to three maximization objectives (each color represents one objective). Edges denote domination relations, i.e., a higher-level node dominates a lower-level node. The node levels correspond to the series of Pareto fronts. The first Pareto front consists of only one solution represented by the top node.

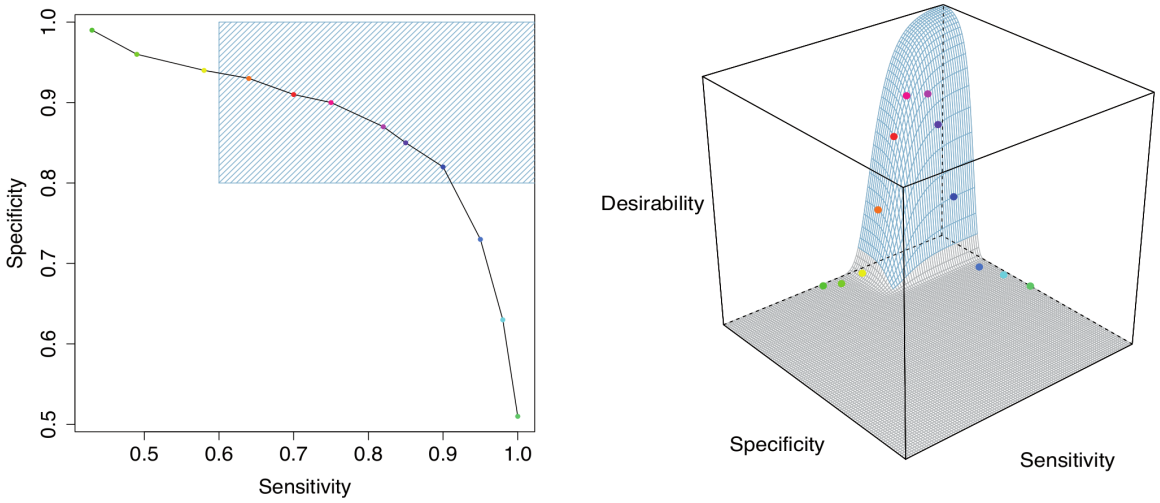

Figure 3: Left: Sensitivity and specificity of a set of Pareto-optimal solutions. In this example, the Pareto front corresponds to a ROC curve. The feasible region (shaded in blue) was restricted to a sensitivity of at least 0.6 and a specificity of 0.8. Right: The same Pareto-optimal solutions embedded in a desirability landscape. The desirability was calculated using two one-sided Harrington functions aggregated by the geometric mean. Similar to the above feasible region, the parameters for the functions were chosen as $\left(y^{(1)} = 0.6, d^{(1)} = 0.01\right)$ and $\left(y^{(2)} = 0.99, d^{(2)} = 0.99\right)$ for the sensitivity and $\left(y^{(1)} = 0.8, d^{(1)} = 0.01\right)$ and $\left(y^{(2)} = 0.99, d^{(2)} = 0.99\right)$ for the specificity. For comparison, the feasible region of the strict clipping at a sensitivity of 0.6 and a specificity of 0.8 is again highlighted in blue.

build their trees at random). This means that the resulting partially ordered set constitutes an approximation of the true ordering.

In some cases, certain extreme trade-offs may be inappropriate. For example, one usually does not want to obtain a classifier with a perfect reclassification error, but a bad generalization performance in cross-validation experiments.

A simple way of handling this is to "clip" the Pareto front to a desired range of trade-offs (Figure 3, left). This is accomplished by specifying upper bounds for minimization objectives or lower bounds for maximization objectives, i.e., restricting the feasible region to a hypercube. In some cases, none of the solutions may be located in the feasible region.

Another way of imposing restrictions on objectives are desirability functions originally proposed by Harrington (1965). Essentially, the approach consists of transforming the objective scores according to their desirability (usually to a range of $[0, 1]$, where a value of 0 means that this score is inappropriate and a value of 1 means that this is a desired value), and of combining them to an overall desirability index, often the geometric mean or the minimum. The desirability transformation can help to add additional properties of utility, e.g., instead of solely using specification limits like in "clipping" these transformations can emphasize notions of mid-specification quality. The transformation also ensures that all objectives operate on a comparable scale. The desirability indices for the Pareto-optimal parameter configurations can then be calculated, and the configurations can be ranked according to their desirability. This usually yields low ranks for the more balanced solutions and high ranks for the configurations in which only one objective has an extreme value. However, unlike in the clipping approach, these extreme configurations are not thrown away. Thus, the desirability ranking approach is a softer way of handling constraints.

A well-known desirability function is Harrington's one-sided desirability function, which realizes a special form of the Gompertz sigmoidal curve (see Harrington 1965; Wagner and Trautmann 2010):

$$d(y) = \exp(-\exp(-b_0 + b_1 \cdot y))),$$

where b_0 and b_1 can be calculated from two tuples of given objective values and the corresponding desirabilities, $(y^{(1)}, d^{(1)})$ and $(y^{(2)}, d^{(2)})$:

$$\begin{aligned} b_0 &= -\log(-\log(d^{(1)})) - b_1 y^{(1)} \\ b_1 &= (-\log(-\log(d^{(2)})) + \log(-\log(d^{(1)})))/(y^{(2)} - y^{(1)}) \end{aligned}$$

In practice, often the corresponding objective values for the desirabilities $d^{(1)} = 0.01$ and $d^{(2)} = 0.99$ are chosen (see Figure 3 for an example).

2.2. Sampling strategies

The input of our method are intervals or lists of values for the parameters to optimize. The algorithm trains a classifier for combinations of parameter values and applies user-defined objective functions, such as the classification error, the sensitivity, or the specificity in reclassification or cross-validation experiments. It returns the Pareto set (or an approximation thereof) which comprises optimal parameter configurations with different trade-offs.

The choice of parameter configurations to rate is a crucial step for the optimization. If all parameters have a (small) finite number of possible values, a full search of all possible

combinations can be performed. In case of continuous parameters or large sets of possible values, sampling strategies have to be applied to draw a subset of n^* parameter configurations.

The most obvious strategy is to simply draw parameter values uniformly at random from the specified ranges or sets. However, this strategy does not ensure that parameter configurations are distributed evenly in the d-dimensional parameter space.

A well-known strategy to ensure a good coverage of the multivariate parameter space is Latin hypercube sampling (McKay *et al.* 1979). This strategy places each parameter range on one side of a d-dimensional hypercube. Continuous parameters are then divided into n^* subintervals, and for each of these intervals, one value is drawn uniformly at random. Discrete parameters are placed on a grid with n^* points such that the difference in the frequencies of any two parameter values on the grid is at most 1. Finally, the n^* values in the d dimensions are joined to form n^* parameter configurations.

Another possibility of covering the parameter space is the use of quasi-random low-discrepancy sequences (Niederreiter 1992). These sequences are designed to distribute points evenly in an interval or hypercube (see, e.g., Maucher *et al.* 2011 for an application). Low discrepancy guarantees that any optimal parameter combination is in close distance to a configuration in the sample. As there is no such guarantee in random strategies such as uniform selection or Latin hypercube sampling, we generally recommend the usage of such sequences rather than these strategies.

We use three multi-dimensional quasi-random sequences:

- The *Halton sequence* in the bases $b_1, ..., b_d$ is defined as $X^H(n) = (\Theta_{b_1}(n), \ldots, \Theta_{b_d}(n))$, where Θ_b is a van der Corput sequence (van der Corput 1935) with base b, i.e.

$$\Theta_b(n) = \sum_{j=0}^{\infty} a_j(n) b^{-j-1} \quad .$$

  Here, the a_j are digits of the base b representation of n, i.e., $n = \sum_{j=0}^{\infty} a_j(n) b^j$ and $a_j \in \{0, 1\}$ for all j.

- A one-dimensional *Sobol sequence* $X^S(n)$ is computed from a sequence of states defined by the recursion

$$m_i = \left(\bigoplus_{j=1}^{s} 2^j m_{i-j} c_j \right) \oplus m_{i-s}$$

  via

$$X^S(n) = \bigoplus_{i=0}^{k-1} a_i(n) m_i 2^{-i-1} \quad ,$$

  where $p = x^s + c_1 x^{s-1} + \ldots + c_{s-1} + 1$ is a primitive polynomial of degree s in the field $\mathbb{Z}_2$ and the a_i denote the binary representation of n. Here, $x \oplus y$ denotes the bitwise XOR of the binary representations of x and y.
  For a multi-dimensional Sobol sequence, one-dimensional Sobol sequences are combined, i.e.

$$X^S(n) = (x_1(n), \ldots, x_d(n))$$

where $x_i(n)$ is the n-th element of a one-dimensional Sobol sequence obtained from a polynomial p_i and all polynomials $p_1, \ldots, p_d$ are pairwise different. For an exemplary implementation, see Bratley and Fox (1988).

- The *Niederreiter sequence* of dimension d and base b is defined as

$$X^N(n) = (x_1(n), \ldots, x_d(n))$$

with

$$x_i(n) = \sum_{j=1}^{\infty} b^{-j} \lambda_{ij} \left(\sum_{r=0}^{\infty} c_{jr}^{(i)} \psi_r(a_r(n-1)) \right),$$

where computations are performed in a commutative ring R with identity and cardinality b, $\psi_r : \{0, 1, \ldots, b-1\} \to R$ and $\lambda_{ij} : R \to \{0, 1, \ldots, b-1\}$ are appropriately chosen bijections and $c_{jr}^{(i)}$ are appropriately chosen elements of R. The a_r denote the digits of the base b representation of n. For more details, e.g., about the choice of the bijections, see Niederreiter (1988) and references cited therein.

Figure 4 shows 100 data points sampled from $[0, 1]^2$ using uniform random numbers (Panel 1), Latin hypercube sampling (Panel 2), and two dimensional quasi-random sequences (Panels 3

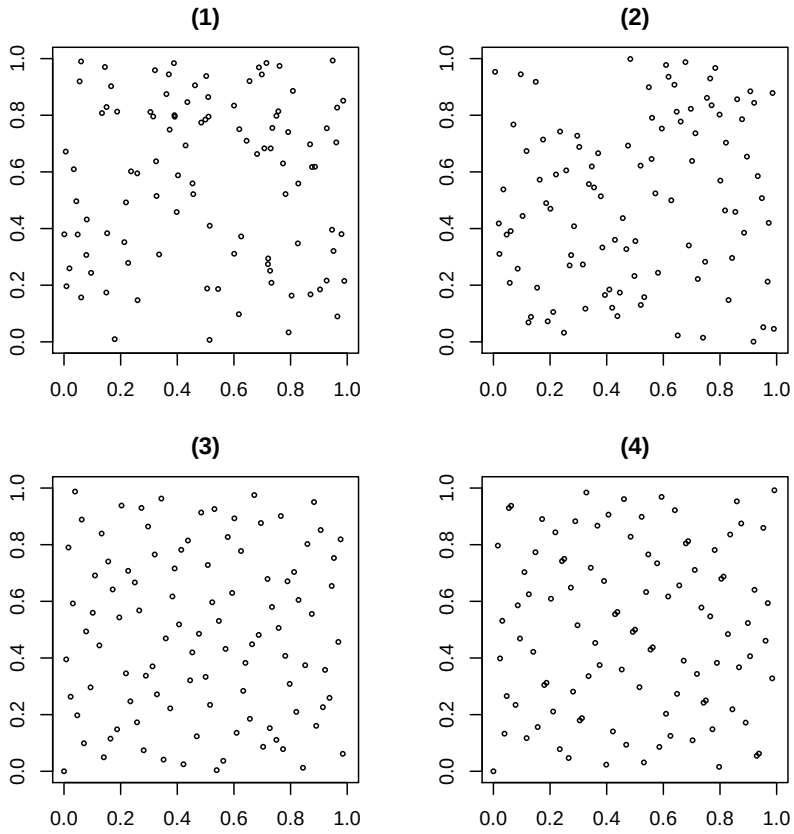

Figure 4: Two-dimensional samples with 100 points chosen according to (1) uniform random numbers, (2) Latin hypercube sampling, (3) a Halton sequence and (4) a Niederreiter sequence.

and 4). In this low-dimensional example, the Niederreiter sequence and the Sobol sequence (Panel 4) are identical. The examples show that quasi-random sequences achieve a more regular coverage of the search space than random numbers and Latin hypercube sampling. In the context of quasi-Monte Carlo integration, Morokoff and Caflisch (1995) showed that Halton sequences generally give better results for up to 6 dimensions, whereas Sobol sequences (and Niederreiter sequences, as they are a generalization of Sobol sequences) perform better for higher dimensions. This may give a rough guidance for choosing the proper sequence.

For very large parameter spaces and in particular for continuous parameters, too many configurations may be required to cover the search space in a sufficient resolution. For such cases, it may be preferable to tune parameters according to a search heuristic. **TunePareto** implements a multi-objective evolutionary algorithm for this purpose. The employed algorithm is based on the well-known NSGA-II approach (Deb *et al.* 2002). This approach employs a multi-objective selection procedure in conjunction with a crowding distance to obtain a good coverage of the Pareto front.

Unlike the original NSGA-II, our implementation introduces some features known from Evolution Strategies (see, e.g., Eiben and Smith 2003; Beyer and Schwefel 2002), in particular a self-adaptation of mutation rates. This reduces the number of parameters of the algorithm, as the distribution parameters for mutation and recombination do not have to be specified. It also allows for a more differentiated mutation scheme, as the genes (i.e., the classifier parameters) are mutated using individual mutation rates. The following briefly characterizes the algorithm implemented in **TunePareto**:

Representation An individual c consists of genes $g_1, \ldots, g_d$ corresponding to the d parameters to optimize. For each continuous parameter g_k, an individual has a mutation rate σ_k.

Initialization The first generation of μ individuals is drawn at random from the parameter space using Latin hypercube sampling.

Fitness measurement The fitness of an individual c is the vector of the M objective function values $F(c_i) = (f_1(c_i), \ldots, f_M(c_i))$. In addition, a crowding distance is assigned to each individual. This crowding distance quantifies the uniqueness of a fitness vector compared to other vectors and preserves diversity on the Pareto front. Let $\mathrm{rk}_m(c_i)$ be the rank of configuration c_i when sorting the configurations according to objective f_m in increasing order. For each objective f_m and each configuration c_i, the cuboid formed by the nearest neighbours with respect to this configuration is

$$d_m(c_i) = \begin{cases} \infty & \mathrm{rk}_m(c_i) = 1 \\ \infty & \mathrm{rk}_m(c_i) = n \\ f_m(c_{\mathrm{rk}_m(c_i)+1}) - f_m(c_{\mathrm{rk}_m(c_i)-1}) & \text{otherwise} \end{cases}$$

Here, we assume that the range of f_m is $[0, 1]$, which means that objective functions with different ranges have to be normalized. The total crowding distance of configuration c_i is then

$$D(c_i) = \sum_{m=1}^{M} d_m(c_i).$$

This crowding distance is employed both for parent selection and survivor selection.

Recombination In each generation, λ offspring are created from the μ parents by randomly mating two parents. For each of the λ offspring, two parents are chosen according to a tournament selection using the crowded-comparison operator $\prec_n$ by Deb *et al.* (2002):

$$c_i \prec_n c_j \text{ if } \begin{cases} c_i \in \mathcal{P}_{l_1}, c_j \in \mathcal{P}_{l_2} \text{ and } l_1 < l_2 \\ c_i, c_j \in \mathcal{P}_l \text{ and } D(c_i) > D(c_j) \end{cases}$$

That is, a configuration c_i is better than a configuration c_j if it is in a better Pareto set than c_j or if it is in the same Pareto set, but has a higher crowding distance. A discrete recombination scheme is used to determine the classifier parameter values of the children: For each parameter, the value of one of the parents is chosen at random. For the mutation rates, intermediate recombination is used (i.e., the mean of the two parent mutation rates is taken for the child). This corresponds to a commonly used recombination scheme in Evolution Strategies.

Mutation Each of the offspring is mutated. For the continuous parameters, we use uncorrelated mutations, i.e.

$$\begin{aligned} \sigma'_k &= \sigma_k \cdot \exp\left(N(0,\tau') + N(0,\tau)\right) \\ g'_k &= g_k + N(0,\sigma'_k) \end{aligned}$$

with $\tau' = 1/\sqrt{2d}$, $\tau = 1/\sqrt{2\sqrt{d}}$ and $N(m,s)$ being a value drawn from the normal distribution with mean m and standard deviation s.

Discrete parameters are mutated with a probability of $\frac{1}{d}$. For integer parameters, mutations are applied by choosing one of $\{g_k - 1, g_k + 1\}$. For nominally scaled parameters, a new value is chosen uniformly at random.

Survivor selection The next generation is selected in a $\mu+\lambda$ strategy by merging the previous generation and the offspring and then applying the non-dominated sorting procedure also used in NSGA-II (Deb *et al.* 2002): The Pareto sets $\mathcal{P}_i$ of the configurations are determined, and parameter configurations are taken from the successive sets (starting with P_1) until the desired generation size μ is reached. If there are more configurations in the current Pareto set than required to obtain the generation size μ, configurations are chosen according to their crowding distances, taking the configurations with the highest crowding distances D.

3. The TunePareto package

At the core of the package is the general function `tunePareto`, which can be configured to select parameters for most standard classification methods provided in R. Classifiers are encapsulated in `TuneParetoClassifier` objects. These objects constitute a way of describing the calls to training and prediction methods of a classifier in a very generic way. **TunePareto** includes predefined comfortable interfaces to frequently used classifiers, i.e., for k-nearest neighbour (k-NN), support vector machines (SVM, Vapnik 1998), decision trees (Breiman *et al.* 1984), random forests (Breiman 2001), and naïve Bayes (Duda and Hart 1973; Domingos and Pazzani 1997). For all other classifiers, such wrappers can be obtained using the `tuneParetoClassifier` function.

Parameters are selected according to one or several objective functions. A set of objective functions are predefined in **TunePareto**, such as the error, sensitivity and specificity, the confusion of two classes in reclassification and cross-validation experiments, or the error variance across several cross-validation runs. Cross-validation experiments can be performed by using a stratified or a non-stratified cross-validation. It is also possible to define custom objective functions, which is supported by various helper functions.

In the following example, we apply a random forest classifier (Breiman 2001) to the Parkinson data set (Little *et al.* 2009) available from the University of California at Irvine (UCI) machine learning repository (Frank and Asuncion 2010) at http://archive.ics.uci.edu/ml/datasets/Parkinsons. This dataset consists of biomedical voice measurements and has 195 samples with 23 features each. 48 of the samples belong to healthy individuals, and 147 belong to patients with Parkinson's disease. The number of trees is optimized according to the average error and the average sensitivity in a 10-fold cross-validation which is repeated 10 times. In this example, we use a stratified cross-validation. A stratified cross-validation ensures that the percentage of samples from a certain class in each fold of the cross-validation corresponds to the percentage of samples in this class in the entire data set. By default, the cross-validation is not stratified. The example uses the **randomForest** package (Liaw and Wiener 2002).

```
R> d <- read.table("parkinsons.data", sep=",", header = TRUE)
R> parkinsons <- d[, colnames(d) != c("name","status")]
R> parkinsons.labs <- d[, "status"]
R> result <- tunePareto(data = parkinsons, labels = parkinsons.labs,
+    classifier = tunePareto.randomForest(), ntree = seq(20, 300, 20),
+    objectiveFunctions = list(
+      cvError(nfold = 10, ntimes = 10, stratified = TRUE),
+      cvSensitivity(nfold = 10, ntimes = 10, stratified = TRUE,
+        caseClass = 1)))
```

`tunePareto` is supplied with the data set, the corresponding class labels, the tuned parameters, and the objective functions. The tuned parameters are supplied in the ... argument (in this case the single parameter `ntree`). Possible values of parameters can either be specified as lists of possible values, or as continuous parameter ranges using the function `as.interval`. From the specified ranges of all optimized parameters, combinations of values are generated and tested. By default, all possible combinations are tested. If one would like to specify a certain set of combinations to be tested, the `parameterCombinations` parameter can be set instead of supplying the value ranges in the ... argument.

Printing the resulting object shows the Pareto-optimal solutions among the tested configurations and their objective values:

```
R> result

Pareto-optimal parameter sets:
              CV.Error CV.Sensitivity
ntree = 220 0.08564103      0.9727891
```

In this case, there are three Pareto-optimal solutions. The objective scores of all (not only the optimal) solutions can be viewed by printing `result$testedObjectiveValues`.

3.1. Sampling strategies

If continuous parameters are used, a full search is not possible. Here, we have to apply a sampling strategy in order to obtain a good coverage of the parameter space. The following example uses Latin hypercube sampling to optimize the `cost` and `gamma` parameters of an RBF support vector machine using 30 samples. The sampling strategy is specified using the `sampleType` parameter. Parameters are tuned according to sensitivity and the mean class-wise cross-validation error (`CV.WeightedError`). This error rate accounts for unbalanced classes as in the Parkinsons data set. As we would like to compare the results of the tuning process with other sampling strategies, we generate the partition for the cross-validation in advance using `generateCVRuns`. This returns a list structure specifying the folds for the different repetitions of the cross-validation. This structure can be supplied to the cross-validation objectives in the `foldList` parameter, ensuring that all experiments are based on the same cross-validation partition.

```
R> foldList <- generateCVRuns(labels = parkinsons.labs, nfold = 10,
+    ntimes = 10, stratified = TRUE)
R> result <- tunePareto(data = parkinsons, labels = parkinsons.labs,
+    classifier = tunePareto.svm(), gamma = as.interval(0.01, 1),
+    cost = as.interval(0.01, 10), kernel = "radial", sampleType = "latin",
+    numCombinations = 30, objectiveFunctions = list(
+      cvWeightedError(foldList = foldList),
+      cvSensitivity(foldList = foldList, caseClass = 1)))
R> result

Pareto-optimal parameter sets:
                                 CV.WeightedError CV.Sensitivity
gamma = 0.44602, cost = 1.7387         0.13112245      0.9877551
gamma = 0.19065, cost = 6.8612         0.08095238      0.9714286
gamma = 0.91161, cost = 7.5325         0.29479167      1.0000000
gamma = 0.31212, cost = 6.5798         0.10710034      0.9816327
gamma = 0.87278, cost = 3.6721         0.27602041      0.9979592
gamma = 0.51687, cost = 5.8632         0.15372024      0.9904762
gamma = 0.082996, cost = 0.87533       0.22670068      0.9965986
gamma = 0.38107, cost = 2.4217         0.11305272      0.9863946
gamma = 0.85638, cost = 5.0885         0.26906888      0.9972789
gamma = 0.54799, cost = 8.5803         0.17285289      0.9938776
...
```

As outlined in the previous section, **TunePareto** also includes sampling strategies based on quasi-random sequences. In the next example, we perform the same type of optimization (with the same cross-validation partitions), but use Halton sequences instead of Latin hypercube sampling.

```
R> result <- tunePareto(data = parkinsons, labels = parkinsons.labs,
+    classifier = tunePareto.svm(), gamma = as.interval(0.01, 1),
+    cost = as.interval(0.01, 10), kernel = "radial", sampleType = "halton",
+    numCombinations = 30, objectiveFunctions = list(
```

```
+      cvWeightedError(foldList = foldList),
+      cvSensitivity(foldList = foldList, caseClass = 1)))
R> result

Pareto-optimal parameter sets:
                              CV.WeightedError CV.Sensitivity
gamma = 0.505, cost = 3.34          0.14849065      0.9884354
gamma = 0.2575, cost = 6.67         0.09145408      0.9795918
gamma = 0.7525, cost = 1.12         0.26281888      0.9972789
gamma = 0.13375, cost = 4.45        0.07954932      0.9700680
gamma = 0.38125, cost = 2.23        0.11513605      0.9863946
gamma = 0.87625, cost = 5.56        0.27914541      0.9979592
gamma = 0.44312, cost = 8.15        0.12625425      0.9870748
gamma = 0.93812, cost = 2.6         0.30104167      1.0000000
gamma = 0.53594, cost = 9.26        0.16311650      0.9925170
gamma = 0.28844, cost = 0.75        0.19128401      0.9965986
...
```

An entirely different way of exploring the parameter space is **TunePareto**'s evolutionary search algorithm as described in Section 3.1. Although the repeated fitness evaluations can be costly, it is advantageous if the parameter space is large. The following example optimizes the SVM parameters using a population of `mu = 20` individuals with `lambda = 20` offspring and 50 generations. With around 1000 fitness evaluations, this example takes 20 minutes on a 2 Intel Xeon CPUs with 3.2 GHz. In real tuning problems, the population size and the number of generations might be increased at the cost of higher computation times. The example also shows how overall computation time can be reduced using parallelization: By setting the parameter `useSnowfall` to `TRUE`, `tunePareto` starts multiple parameter evaluations in parallel on a previously initialized **snowfall** cluster (Knaus *et al.* 2009). This allows for both multicore and network computing.

```
R> library("snowfall")
R> sfInit(parallel = TRUE, cpus = 2, type = "SOCK")
R> result <- tunePareto(data = parkinsons, labels = parkinsons.labs,
+    classifier = tunePareto.svm(), gamma = as.interval(0.01, 1),
+    cost = as.interval(0.01, 10), kernel = "radial",
+    sampleType = "evolution", numIterations = 50, mu = 20, lambda = 20,
+    objectiveFunctions = list(cvWeightedError(foldList = foldList),
+      cvSensitivity(foldList = foldList, caseClass = 1)),
+    useSnowfall = TRUE)
R> sfStop()
R> result

Pareto-optimal parameter sets:
                               CV.WeightedError CV.Sensitivity
gamma = 0.91082, cost = 9.4551        0.29479167      1.0000000
gamma = 0.12305, cost = 9.0791        0.07189626      0.9687075
gamma = 0.90369, cost = 9.5719        0.29200680      0.9993197
```

```
gamma = 0.01, cost = 8.487          0.21005527      0.9986395
gamma = 0.011022, cost = 8.2169     0.20899235      0.9965986
gamma = 0.54952, cost = 4.7994      0.17285289      0.9938776
gamma = 0.51226, cost = 9.4185      0.15197704      0.9897959
gamma = 0.42367, cost = 10          0.12312925      0.9870748
gamma = 0.35955, cost = 9.1541      0.10888605      0.9863946
gamma = 0.27508, cost = 10          0.10153061      0.9802721
...
```

The three approaches yield mean class-wise errors (`CV.WeightedError`) ranging from 0.07 to 0.3 and sensitivity ranging from 0.97 to 1.0. The full results show that Latin hypercube sampling and Halton sequences are outperformed by the evolutionary search in this case (some optimal configurations were omitted here for readability). The joint Pareto front of the three examples (calculated using the `mergeTuneParetoResults` function) contains 4 configuration determined by Latin hypercube sampling and 5 configurations from Halton sampling, but 18 configurations from the evolutionary approach. As mentioned before, these results are subject to many stochastic factors, so that this may not be a general statement.

3.2. Visualization

A classical way of visualizing the results of a multi-objective optimization is plotting the (approximated) Pareto fronts. In **TunePareto**, this is accomplished using the `plotParetoFronts2D` function. For the above results

```
R> plotParetoFronts2D(result, drawLabels = FALSE)
```

plots the 2-dimensional Pareto front. To enhance clarity, the labels of the points (i.e., the parameter values) are suppressed.

Figure 5 shows the approximated Pareto fronts for the three examples above, using Latin Hhypercube sampling, Halton sequences and the evolutionary search for the parameter selection. Here, the first Pareto front (the blue line) corresponds to the above results. For

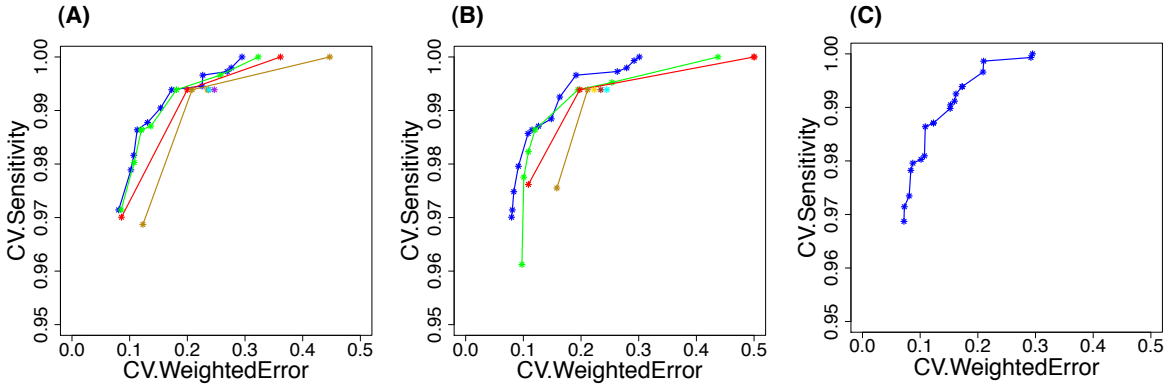

Figure 5: Panel A: Pareto fronts for the optimization of the `cost` and `gamma` parameters of an SVM according to sensitivity and the mean class-wise error using Latin hypercube sampling. Panel B: Pareto fronts of the same optimization using Halton sequences. Panel C: Pareto fronts of the same optimization using evolutionary search.

the evolutionary algorithm, all returned solutions are Pareto-optimal, so that there is only a single front.

If parameters are selected according to more than two objectives, the standard 2-dimensional plot is not applicable. **TunePareto** includes two further plots that can cope with more than two objectives. In the following example, we optimize an SVM according to the cross-validation error, the sensitivity and the specificity with respect to class 1. We use a sampling strategy according to the Niederreiter sequence. This requires the **gsl** package (Hankin 2006), a wrapper for the GNU Scientific Library.

```
R> result <- tunePareto(data = parkinsons, labels = parkinsons.labs,
+    classifier = tunePareto.svm(), gamma = as.interval(0.01, 1),
+    cost = 1, kernel = "radial", sampleType = "niederreiter",
+    numCombinations = 20, objectiveFunctions = list(
+      cvError(nfold = 10, ntimes = 10),
+      cvSensitivity(nfold = 10, ntimes = 10, caseClass = 1),
+      cvSpecificity(nfold = 10, ntimes = 10, caseClass = 1)))
R> result

Pareto-optimal parameter sets:
                  CV.Error CV.Sensitivity CV.Specificity
gamma = 0.43346 0.09589744      0.9918367      0.6354167
gamma = 0.68096 0.13589744      0.9979592      0.4541667
gamma = 0.18596 0.07538462      0.9904762      0.7229167
gamma = 0.74283 0.14564103      1.0000000      0.4083333
gamma = 0.49533 0.10102564      0.9938776      0.6083333
gamma = 0.61908 0.11846154      0.9959184      0.5312500
gamma = 0.58814 0.11025641      0.9952381      0.5666667
```

The results are depicted in a matrix plot in Figure 6: For each pair of objectives, the approximated Pareto fronts are plotted as in the above example, and the plots are accumulated in a matrix structure. This makes a visual comparison of more than 2 objectives possible. Pairs of objectives always occur twice in the matrix – each objective is plotted once on each axis. The labels correspond to the configurations. By default, labels are drawn in such a way that they do not overlap with each other and do not exceed the margins of the plot, which means that some labels may be omitted. All labels can be drawn by setting `fitLabels = FALSE`. Although the Pareto fronts of the single 2-dimensional plots in the matrix do not correspond to the overall Pareto front, this pairwise comparison can reveal relations of objectives that are not visible from the overall results. For example, the top-right and bottom-left plots reveal that in this case, the cross-validation error and the specificity are not concurrent. We gain a total ordering on both objectives as each Pareto front consists only of a single configuration. This means that it might suffice to optimize only one objective. Recalculating the Pareto set according to sensitivity and specificity – omitting the cross-validation error – highlights this: The Pareto-optimal solutions are the same as in the above example using all three objectives.

```
R> subres <- recalculateParetoSet(result, objectives = c(2, 3))
R> subres
```

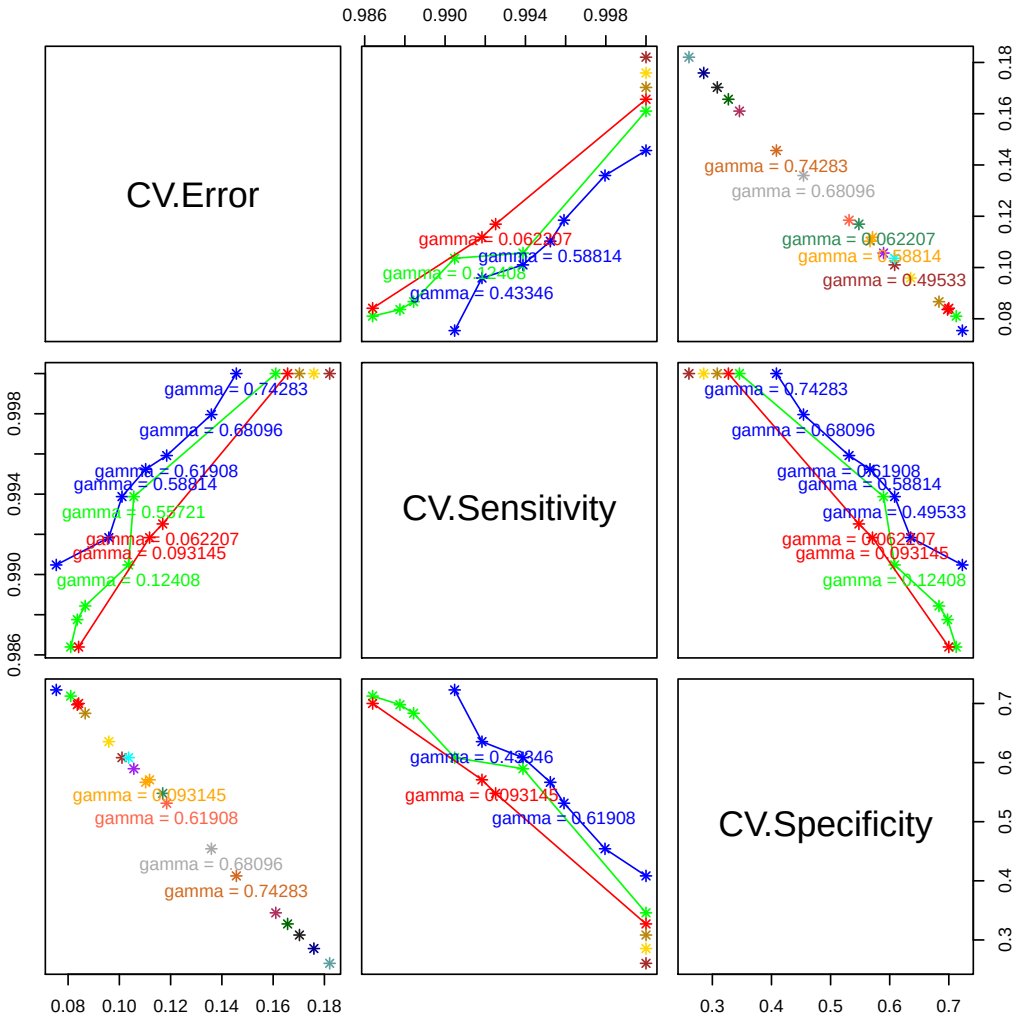

Figure 6: A matrix of Pareto front plots for all pairs of objectives in a 3-objective optimization of the cost parameter of a SVM.

```
Pareto-optimal parameter sets:
                CV.Sensitivity CV.Specificity
gamma = 0.43346      0.9918367      0.6354167
gamma = 0.68096      0.9979592      0.4541667
gamma = 0.18596      0.9904762      0.7229167
gamma = 0.74283      1.0000000      0.4083333
gamma = 0.49533      0.9938776      0.6083333
gamma = 0.61908      0.9959184      0.5312500
gamma = 0.58814      0.9952381      0.5666667
```

A further possibility to visualize optimizations with more than two objectives is to plot the Pareto front approximations in a graph (see Figure 7). When read from left to right, this

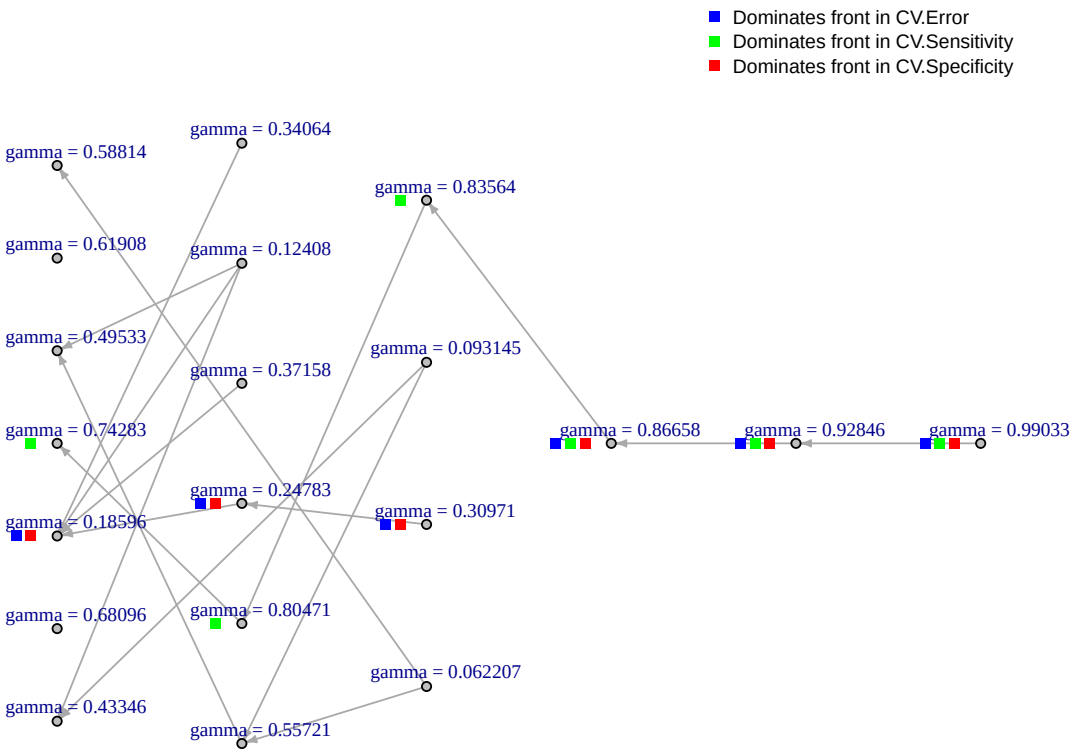

Figure 7: A domination graph of the SVM cost parameter. This corresponds to a rotated Hasse diagram with additional color indicators (see Figure 2). The nodes represent the parameter configurations and are ordered in columns according to the Pareto fronts. The edges represent dominance relations between two configurations. For example, `gamma = 0.12408` is dominated by `gamma = 0.49533`. The color indicators show in which objective a configuration is optimal with respect to its Pareto front (e.g., `gamma = 0.74283` has the best sensitivity in the first Pareto front).

corresponds to a Hasse diagram of the dominance relations with an additional color encoding for the best values in an objective (see also Figure 2).

```
R> plotDominationGraph(result, legend.x = "topright")
```

The function `plotDominationGraph` is based on the **igraph** package (Csardi and Nepusz 2006). Each node in the graph corresponds to one parameter configuration, and an edge corresponds to a dominance relation. The nodes are ordered such that the columns correspond to Pareto fronts. Small color indicators next to the nodes show in which of the objectives the corresponding configuration is optimal with respect to its Pareto front. In the default setting, transitive dominance relations are not drawn, as they are always caused by multiple direct dominance relations of configurations. Transitive edges can be included by setting the parameter `transitiveReduction` to `FALSE`. This is a more abstract representation than the usual Pareto front plot, as the actual scores for the objectives are not depicted. The graph representation allows for capturing dominance relations among the configurations at a glance and is suitable for any number of dimensions.

3.3. Selecting configurations

In principle, all solutions on the (first) Pareto front can be viewed as equally good. However, there are often additional requirements for the solutions. Consider an example similar to the one above: We tune the SVM `gamma` parameter according to specificity and sensitivity.

```
R> result1 <- tunePareto(data = parkinsons, labels = parkinsons.labs,
+    classifier = tunePareto.svm(), gamma = as.interval(0.01, 1),
+    cost = 1, kernel = "radial", sampleType = "niederreiter",
+    numCombinations = 20, objectiveFunctions = list(
+      cvSensitivity(nfold = 10, ntimes = 10, caseClass = 1),
+      cvSpecificity(nfold = 10, ntimes = 10,  caseClass = 1)))
R> result1

Pareto-optimal parameter sets:
                CV.Sensitivity CV.Specificity
gamma = 0.43346      0.9918367      0.6354167
gamma = 0.68096      0.9979592      0.4541667
gamma = 0.18596      0.9904762      0.7229167
gamma = 0.74283      1.0000000      0.4083333
gamma = 0.49533      0.9938776      0.6083333
gamma = 0.61908      0.9959184      0.5312500
gamma = 0.58814      0.9952381      0.5666667
```

The resulting set of optimal configurations comprises some configurations with very extreme trade-offs. For example, it is possible to obtain a perfect sensitivity of 1, but at the cost of a low specificity of only 0.4. This is often not desirable. Suppose we would like to rule out solutions with a specificity or sensitivity below 0.6. We can specify this directly as boundaries in the optimization using the `objectiveBoundaries` parameter.

```
R> result2 <- tunePareto(data = parkinsons, labels = parkinsons.labs,
+    classifier = tunePareto.svm(), gamma = as.interval(0.01, 1),
+    cost = 1, kernel = "radial", sampleType = "niederreiter",
+    numCombinations = 20, objectiveFunctions = list(
+      cvSensitivity(nfold = 10, ntimes = 10, caseClass = 1),
+      cvSpecificity(nfold = 10, ntimes = 10, caseClass = 1)),
+    objectiveBoundaries = c(0.6, 0.6))
```

The new result now excludes 4 solutions:

```
R> result2

Pareto-optimal parameter sets matching the objective restrictions:
                CV.Sensitivity CV.Specificity
gamma = 0.43346      0.9918367      0.6354167
gamma = 0.18596      0.9904762      0.7229167
gamma = 0.49533      0.9938776      0.6083333
```

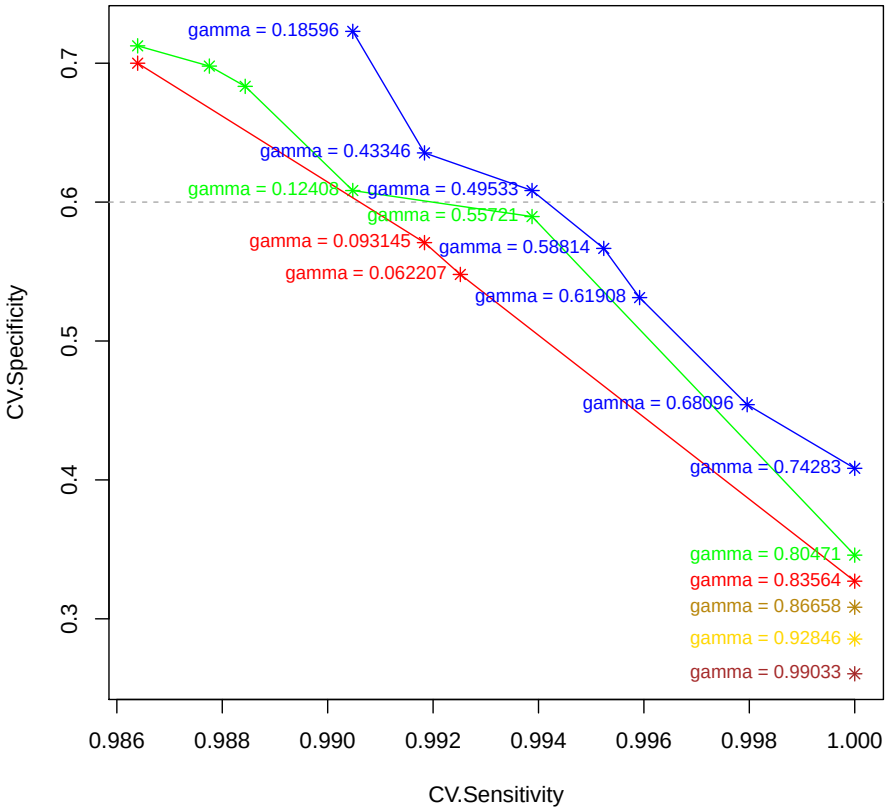

Figure 8: The results of a two-objective optimization of the `gamma` parameter of a SVM with restricted objective values. The valid optimal objective values are located in the upper part (dashed line). The points on the approximated Pareto front outside this region are not considered as being optimal.

We can visualize this using `plotParetoFronts2D`, as depicted in Figure 8. The boundaries are drawn as grey dashed lines. In this case, only the boundary of the specificity is visible, as the boundary for the sensitivity is very far apart from the performance of all solutions and thus outside the drawing region.

Desirability functions constitute another possibility of imposing restrictions to the objective values. The **desire** package (Trautmann *et al.* 2009) provides functions to calculate different kinds of desirability indices. **TunePareto** includes the function `rankByDesirability` ranking the results of a call to `tunePareto` according to such indices:

```
R> library("desire")
R> d1 <- harrington1(y1 = 0.6, d1 = 0.01, y2 = 0.99, d2 = 0.99)
R> d2 <- harrington1(y1 = 0.6, d1 = 0.01, y2 = 0.99, d2 = 0.99)
R> di <- geometricDI(d1, d2)
R> rankByDesirability(result1, di)
```

```
                CV.Sensitivity CV.Specificity Desirability
gamma = 0.18596      0.9904762      0.7229167 7.126124e-01
gamma = 0.43346      0.9918367      0.6354167 2.658462e-01
gamma = 0.49533      0.9938776      0.6083333 1.320273e-01
gamma = 0.58814      0.9952381      0.5666667 2.040430e-02
gamma = 0.61908      0.9959184      0.5312500 1.129601e-03
gamma = 0.68096      0.9979592      0.4541667 1.291991e-10
gamma = 0.74283      1.0000000      0.4083333 4.835754e-21
```

In this example, the objective values are rated according to Harrington's one-sided desirability function (Harrington 1965). Again, we set a value of 0.6 as a margin for both specificity and sensitivity. A value of 0.99 is considered as nearly optimal. The desirability functions of a parameter configuration c are aggregated according to the geometric mean. The values of the desirability index `di` are used to rank the Pareto-optimal configurations. The example shows that Pareto-optimal solutions with balanced objective values are ranked higher than those with an extremely good performance in a single objective. This behaviour is influenced by the choice of the of the geometric mean for the desirability index and may change when using different desirability indices. Here, solutions with a better specificity are preferred, as the sensitivity is always close to the maximum.

3.4. Customizing TunePareto

The **TunePareto** package is flexible and can be extended by custom classifier wrappers and objective functions.

Classifiers are encapsulated in `TuneParetoClassifier` objects, which describe the calls needed for training and applying a classifier in `TunePareto`. To utilize these methods directly, `TuneParetoClassifier` objects can not only be used in `tunePareto`, but also in special training and prediction functions (`trainTuneParetoClassifier` and `predict.TuneParetoModel`) that can be integrated into other custom tuning procedures.

We use the random forest classifier to illustrate the creation of custom classifier objects:

```
R> forest <- tuneParetoClassifier(name = "randomForest",
+    classifier = randomForest, predictor = predict,
+    classifierParamNames = "ntree", predictorParamNames =  NULL,
+    useFormula = FALSE, trainDataName = "x", trainLabelName = "y",
+    testDataName = "newdata", modelName = "object",
+    requiredPackages = "randomForest")
```

The `tuneParetoClassifier` function creates a wrapper for the classifier to be called. The `name` parameter specifies a human-readable name of the classifier. The further parameters specify the type and arguments of the classifier and predictor methods. Here, `classifier` specifies the classifier training function, and `predictor` specifies the prediction function. It is also possible to call a function that integrates both training and prediction by leaving the `predictor` parameter empty. `classifierParamNames` and `predictorParamNames` are vectors that define the names of arguments that are accepted as valid parameters for the classifier and the predictor function by `tunePareto`. In this case, we specify only the `ntree` parameter, which is the parameter we would like to optimize. Default values for parameters can be set using two further parameters `predefinedClassifierParams` and `predefinedPredictorParams`.

`trainDataName`, `trainLabelName`, `testDataName` and `modelName` are string parameters that specify the names of the arguments of the training and prediction functions for the training data, the class labels, the test data for prediction, and the trained model in the prediction function respectively. The `requiredPackages` parameter lists the packages that are required to run the classifier. These packages are loaded automatically by **TunePareto**. If run in a **snowfall** cluster, the packages are loaded on all nodes. The `forest` object resulting from the call can be passed to the `classifier` parameter of `tunePareto`.

The `randomForest` classifier can be called in two ways: by providing the data and the labels using the `x` and the `y` parameters, or by providing a formula and a data frame. `TuneParetoClassifier` wrappers are able to call both interfaces: If a classifier uses a formula interface, we set `useFormula=TRUE`. In this case, `tunePareto` automatically constructs a formula of the form `Class~.` to train a classifier that associates the class labels to all supplied features. The name of the argument that receives the formula can be specified in `formulaName`. The following example is equivalent to the above example, but uses the formula interface of `randomForest`:

```
R> forest <- tuneParetoClassifier(name = "randomForest",
+    classifier = randomForest, predictor = predict,
+    classifierParamNames = "ntree", predictorParamNames =  NULL,
+    useFormula = TRUE, formulaName = "formula", trainDataName = "data",
+    testDataName = "newdata", modelName = "object",
+    requiredPackages = "randomForest")
```

Besides using customized classifiers, it is also possible to introduce user-defined objective functions. In the following example, we define a new objective calculating the false positive rate in a cross-validation:

```
R> cvFalsePositives <- function(nfold = 10, ntimes = 10,
+    leaveOneOut = FALSE, stratified = TRUE, foldList = NULL, caseClass)
+  {
+    return(createObjective(precalculationFunction = "crossValidation",
+      precalculationParams = list(nfold = nfold, ntimes = ntimes,
+        leaveOneOut = leaveOneOut, foldList = foldList,
+        stratified = stratified),
+      objectiveFunction = function(result, caseClass) {
+        return(mean(sapply(result, function(run) {
+          predictedLabels <- unlist(lapply(run, function(fold)
+            fold$predictedLabels))
+          trueLabels <- unlist(lapply(run, function(fold)
+            fold$trueLabels))
+          return(sum(predictedLabels == caseClass &
+            trueLabels != caseClass))
+        })))
+      },
+      objectiveFunctionParams = list(caseClass = caseClass),
+      direction = "minimize", name = "CV.FalsePositives"))
+  }
```

Objective functions are assembled using the `createObjective` function. A good way of encapsulating them is writing a custom function that returns the `TuneParetoObjective` object with the correct setting of additional parameters. This is shown above. `tunePareto` merges common calculations of the objective functions intelligently. This ensures that objectives use the same experimental results and also reduces computation times. For example, if two objectives are calculated from cross-validation results, only one cross-validation is performed. This is achieved by splitting up the objective calculation into two parts: A common precalculation, such as determining the class label predictions in a cross-validation, and the calculation of the score itself, e.g., the determination of the misclassification rate from the class labels.

In the above example, we use the `crossValidation` function as a precalculation function. Precalculation functions receive the classifier, the training and test data, and the parameters as an input. Furthermore, they can take additional parameters defined in the `precalculationParams` argument of `createObjective`. In this case, these are the number of runs and folds, the switches for leave-one-out cross-validation and stratification, and the `foldList` parameter which can be used to supply a precalculated cross-validation partition instead of generating a random partition. The output of a precalculation function is not predefined – it is a single object which is passed directly to the actual scoring functions and can take any form these functions are able to process. Here, it is a list of runs, each containing a list of folds with a vector of true labels and the predicted labels. This list is the first parameter of the function defined in the `objectiveFunction` argument. Like the precalculation function, this scoring function can also have additional parameters. These are specified in the `objectiveFunctionParams` argument. In this case, we have to specify the class which is considered as the positive class for the calculation of the rate (`caseClass`). The scoring function determines the mean fraction of false positive predictions across the runs of the cross-validation. The `direction` argument specifies whether the optimal score is the minimum or the maximum. Furthermore, a readable name is supplied for the objective. The function `cvFalsePositives` can now be called and passed to the `objectiveFunctions` parameter of `tunePareto` just like the objective functions known from the above examples.

4. Discussion

Parameter tuning is an every-day issue for many researchers in the field of machine learning. Parameters are often specified according to rules of thumb and intuition or by rudimentary trials. Automatic parameter tuning has been studied mainly focusing on single classifiers and single objectives.

Parameter selection for classifiers should obey certain standards. Many published results are over-optimistic because the same data is used both for parameter selection and validation of the final classifier. To obtain unbiased results, Bishop (1995) suggests splitting the data into a training set, a validation set, and a test set. The training set and the validation set are used to determine the parameters of the classifiers. The performance of the classifier is then assessed independently on the test set. A similar approach was recently proposed by Boulesteix and Strobl (2009). Varma and Simon (2006) suggest a nested cross-validation for the parameter selection. Bischl *et al.* (2010) describe common pitfalls in the context of tuning and resampling experiments.

TunePareto provides a general framework for the selection of classifier parameters accord-

ing to multiple objectives. Parameter values can be chosen according to intelligent sampling strategies and search heuristics, such as quasi-random sequences and evolutionary algorithms. The package includes wrappers for many state-of-the-art classifiers and objective functions, but can be extended for almost any classifier using arbitrary objective functions. The multi-objective view on the parameter selection problem can help discovering trade-offs of objectives that remain invisible when optimizing according to a single objective. The basic idea is not to determine a single best parameter configuration, but to offer a range of good parameter configuration with different classifier properties, leaving the ultimate decision to the researcher. Decision support is provided by visualization functions. In particular, the package introduces visualization techniques for more than two objectives.

Although it is often advisable to consider several objectives separately, one should keep the number of objectives small. With four or more objectives, so-called many-objective optimization problems arise, which impose additional problems on the tuning process (see, e.g., Ishibuchi *et al.* 2008). In particular, almost all solutions are non-dominated if too many objectives are specified, which means that it is hard to determine the desired solutions. Furthermore, the number of solutions needed to approximate the Pareto front increases exponentially with the number of objective functions.

The stochasticity of the tuning procedure – caused by randomized processes such as the selection of a partition for the cross-validation, random factors in classifiers, and the sampling of parameter combinations – may require to take additional measures. For example, repetitions in the calculation of the objective functions (such as specifying multiple runs in the `ntimes` parameter of cross-validation objectives) can reduce the effects of outliers. Another option is to run the complete tuning process repeatedly and to calculate a joint Pareto front. In particular, the evolutionary search process can benefit from restarts, as it may converge to different optima depending on its initialization and random seed. When merging results from repeated subsampling experiments, one should ensure that all these experiments use the same partitions (e.g., by supplying a pregenerated fold list to a cross-validation) to make the results comparable.

Acknowledgments

This work is supported by the Graduate School of Mathematical Analysis of Evolution, Information and Complexity at the University of Ulm (CM, HAK) and by the German Federal Ministry of Education and Research (BMBF) within the framework of the program of Medical Genome Research (PaCa-Net; project ID PKB-01GS08) and Gerontosys (Forschungskern SyStaR). The responsibility for the content lies exclusively with the authors.

Christoph Müssel and Ludwig Lausser contributed equally. Correspondence should be addressed to Hans A. Kestler.

References

Bartz-Beielstein T (2006). *Experimental Research in Evolutionary Computation – The New Experimentalism*. Springer-Verlag, Heidelberg.

Bartz-Beielstein T, Lasarczyk CWG, Preuss M (2005). "Sequential Parameter Optimization."

In *Proceedings of the 2005 Congress on Evolutionary Computation*, volume 1, pp. 773–780. IEEE Press, Piscataway.

Bartz-Beielstein T, Ziegenhirt J, Konen W, Flasch O, Koch P, Zaefferer M (2011). ***SPOT**: Sequential Parameter Optimization*. R package version 0.1.1375, URL http://CRAN.R-project.org/package=SPOT.

Belton V, Stewart TJ (2002). *Multiple Criteria Decision Analysis: An Integrated Approach*. Kluwer Academic Publishers, Dordrecht.

Beyer HG, Schwefel HP (2002). "Evolution Strategies – A Comprehensive Introduction." *Natural Computing*, **1**(1), 3–52.

Bischl B, Mersmann O, Trautmann H (2010). "Resampling Methods in Model Validation." In *Proceedings of the Workshop on Experimental Methods for the Assessment of Computational Systems (WEMACS 2010)*.

Bishop CM (1995). *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford.

Boulesteix AL, Strobl C (2009). "Optimal Classifier Selection and Negative Bias in Error Rate Estimation: An Empirical Study on High-Dimensional Prediction." *BMC Medical Research Methodology*, **9**, 85.

Bratley P, Fox BL (1988). "Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator." *ACM Transactions on Mathematical Software*, **14**, 88–100.

Breiman L (2001). "Random Forests." *Machine Learning*, **45**(1), 5–32.

Breiman L, Friedman J, Stone CJ, Olshen RA (1984). *Classification and Regression Trees*. Chapman & Hall/CRC, Boca Raton.

Chang CC, Lin CJ (2001). ***LIBSVM**: A Library for Support Vector Machines*. URL http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Chapelle O, Vapnik V, Bousquet O, Mukherjee S (2002). "Choosing Multiple Parameters for Support Vector Machines." *Machine Learning*, **46**(1), 131–159.

Chunhong Z, Licheng J (2004). "Automatic Parameters Selection for SVM Based on GA." In *Proceedings of the Fifth World Congress on Intelligent Control and Automation*, pp. 1869–1872. IEEE, Piscataway.

Csardi G, Nepusz T (2006). "The **igraph** Software Package for Complex Network Research." *InterJournal*, **Complex Systems**, 1695. URL http://igraph.sf.net/.

de Souza BF, de Carvalho ACPLF, Calvo R, Ishii RP (2006). "Multiclass SVM Model Selection Using Particle Swarm Optimization." In *Proceedings of the Sixth International Conference on Hybrid Intelligent Systems*, p. 31. IEEE Computer Society, Washington, DC.

Deb K (2004). *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, New York.

Deb K, Pratap A, Agarwal S, Meyarivan T (2002). "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II." *IEEE Transactions on Evolutionary Computation*, **6**(2), 182–197.

Dimitriadou E, Hornik K, Leisch F, Meyer D, Weingessel A (2010). ***e1071**: Misc Functions of the Department of Statistics (e1071), TU Wien*. R package version 1.5-24, URL http://CRAN.R-project.org/package=e1071.

Domingos P, Pazzani M (1997). "On the Optimality of the Simple Bayesian Classifier under Zero-One Loss." *Machine Learning*, **29**, 103–130.

Duda HO, Hart PE (1973). *Pattern Classification and Scene Analysis*. John Wiley & Sons, New York.

Eiben A, Smith J (2003). *Introduction to Evolutionary Computing*. Springer-Verlag, Heidelberg.

Frank A, Asuncion A (2010). "UCI Machine Learning Repository." URL http://archive.ics.uci.edu/ml/.

Fröhlich H, Zell A (2005). "Efficient Parameter Selection for Support Vector Machines in Classification and Regression via Model-Based Global Optimization." In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, volume 3, pp. 1431–1436.

Hankin RKS (2006). "Special Functions in R: Introducing the **gsl** Package." *R News*, **6**, 24–26. URL http://CRAN.R-project.org/doc/Rnews/.

Harrington J (1965). "The Desirability Function." *Industrial Quality Control*, **21**(10), 494–498.

Igel C (2005). "Multi-Objective Model Selection for Support Vector Machines." In *Third International Conference on Evolutionary Multi-Criterion Optimization*, pp. 534–546. Springer-Verlag, Heidelberg.

Ishibuchi H, Tsukamoto N, Nojima Y (2008). "Evolutionary Many-Objective Optimization: A Short Review." In *Proceedings of the 2008 IEEE Congress on Evolutionary Computation*, pp. 2424–2431. IEEE, Piscataway, NJ, USA.

Kalos A (2005). "Automatic Neural Network Structure Determination via Discrete Particle Swarm Optimization (for Non-Linear Time Series Models)." In *Proceedings of the Fifth WSEAS International Conference on Simulation, Modeling and Optimization*.

Kapp MN, Sabourin R, Maupin P (2009). "A PSO-Based Framework for Dynamic SVM Model Selection." In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pp. 1227–1234. ACM, New York.

Knaus J, Porzelius C, Binder H, Schwarzer G (2009). "Easier Parallel Computing in R with **snowfall** and **sfCluster**." *The R Journal*, **1**, 54–59.

Kohavi R, John GH (1995). "Automatic Parameter Selection by Minimizing Estimated Error." In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 304–312. Morgan Kaufmann, San Francisco.

Laux H (2005). *Entscheidungstheorie*. 6th edition. Springer-Verlag, Berlin.

Liaw A, Wiener M (2002). "Classification and Regression by **randomForest**." *R News*, **2**(3), 18–22. URL http://CRAN.R-project.org/doc/Rnews/.

Little MA, McSharry PE, Hunter EJ, Spielman J, Ramig LO (2009). "Suitability of Dysphonia Measurements for Telemonitoring of Parkinson's Disease." *IEEE Transactions on Biomedical Engineering*, **56**(4).

Luc DT (2008). "Pareto Optimality." In A Chinchuluun, PM Pardalos, A Migdalas, L Pitsoulis (eds.), *Pareto Optimality, Game Theory and Equilibria*, pp. 481–515. Springer-Verlag, New York.

Maucher M, Schöning U, Kestler HA (2011). "Search Heuristics and the Influence of Non-Perfect Randomness: Examining Genetic Algorithms and Simulated Annealing." *Computational Statistics*, **26**(2), 303–319.

McKay MD, Beckman RJ, Conover WJ (1979). "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code." *Technometrics*, **21**(2), 239–245.

Morokoff WJ, Caflisch RE (1995). "Quasi-Monte Carlo Integration." *Journal of Computational Physics*, **122**, 218–230.

Niederreiter H (1988). "Low-Discrepancy and Low-Dispersion Sequences." *Journal of Number Theory*, **30**(1), 51–70.

Niederreiter H (1992). *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial Mathematics, Philadelphia.

Pappalardo M (2008). "Multiobjective Optimization: A Brief Overview." In A Chinchuluun, PM Pardalos, A Migdalas, L Pitsoulis (eds.), *Pareto Optimality, Game Theory and Equilibria*, pp. 517–528. Springer-Verlag, New York.

Pontil M, Verri A (1998). "Properties of Support Vector Machines." *Neural Computation*, **10**(4), 955–974.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Suttorp T, Igel C (2006). "Multi-Objective Optimization of Support Vector Machines." In *Multi-Objective Machine Learning*, volume 16 of *Studies in Computational Intelligence*, pp. 199–220. Heidelberg.

Trautmann H, Steuer D, Mersmann O (2009). ***desire**: Desirability Functions*. R package version 1.0.5, URL http://CRAN.R-project.org/package=desire.

van der Corput JG (1935). "Verteilungsfunktionen I." *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, **38**, 813–821.

Vapnik V (1998). *Statistical Learning Theory*. John Wiley & Sons, New York.

Varma S, Simon R (2006). "Bias in Error Estimation When Using Cross-Validation for Model Selection." *BMC Bioinformatics*, **7**, 91.

Wagner T, Trautmann H (2010). "Integration of Preferences in Hypervolume-Based Multiobjective Evolutionary Algorithms by Means of Desirability Functions." *IEEE Transactions on Evolutionary Computation*, **14**(5), 688–701.

Zhang Q, Shan G, Duan X, Zhang Z (2009). "Parameters Optimization of Support Vector Machine Based on Simulated Annealing and Genetic Algorithm." In *Proceedings of the 2009 IEEE International Conference on Robotics and Biomemetics*, pp. 1203–1306.

Zhang Y (2008). "Evolutionary Computation Based Automatic SVM Model Selection." In *Proceedings of the 2008 Fourth International Conference on Natural Computation*, volume 2, pp. 66–70.

Affiliation:

Hans A. Kestler
Research group Bioinformatics and Systems Biology
Institute of Neural Information Processing
University of Ulm
89069 Ulm, Germany
E-mail: hans.kestler@uni-ulm.de
Telephone: +49/731/5024248
Fax: +49/731/5024156

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 46, Issue 5 *Submitted:* 2010-08-17
January 2012 *Accepted:* 2011-10-25